Diplomarbeit

Taclets und *Rewriting Logic*: Bezüge zwischen Java Semantiken

Deutsche Zusammenfassung

von cand. inform.
Ralf Sasse

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Wolfgang Ahrendt, Andreas Roth

Institut für Logik, Komplexität und Deduktionssysteme
Fakultät für Informatik
Universität Karlsruhe

28. April 2005

Erklärung

Ich erkläre hiermit, diese Arbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 28. April 2005

Ralf Sasse

Danksagung

Hier möchte ich mich bei all jenen bedanken, die mich während der Arbeit an dieser Diplomarbeit unterstützt haben. Zuerst möchte ich mich bei Dr. Wolfgang Ahrendt für das Thema und den Anstoss dieser Arbeit sowie die Betreuung bedanken. Ein Dank geht auch an Professor Peter Schmitt dafür, dass er diese Diplomarbeit mit Betreuung ausserhalb der Universität Karlsruhe ermöglicht hat. Vielen Dank an Andreas Roth, der mein Betreuer vor Ort war und viele hilfreiche Bemerkungen zu meiner Arbeit und Implementierung hatte. Ich bin Steffen Schlager, Richard Bubel und Philipp Rümmer für ihre Hilfe, was KeY und Java angeht, dankbar. Ich möchte mich zudem noch bei Professor Reiner Hähnle und Professor Peter Schmitt bedanken, dass sie meinen Aufenthalt in Göteborg ermöglicht haben. Zum Abschluss noch vielen Dank an meine Eltern für ihre Unterstützung meines gesamten Studiums.

Inhaltsverzeichnis

1 Einführung

In dieser Arbeit geht es um das Problem, die Regeln eines Beweiskalküls für Programmiersprachen formal zu validieren. Dies ist wichtig, um zu belegen, dass Beweise die diesen Kalkül nutzen, korrekt sind. Andernfalls gelten alle Beweise nur relativ zur Korrektheit des Kalküls, d.h. sie könnten im ungünstigsten Fall ungültig sein. Wir gehen das Problem mittels einer Vergleichsprüfung der Regeln gegenüber einer Semantik für die Programmiersprache von Interesse an.

Die uns interessierende Programmiersprache ist *Java*, und die Semantik die wir verwenden ist in *Rewriting Logic* gegeben. Wir wollen die Regeln des Kalküls von *KeY*, die als *Taclets* implementiert sind, validieren. Dafür nutzen wir eine in Rewriting Logic gegebene Semantik von Java, welche in *Maude* implementiert ist.

Der zentrale Beitrag unserer Arbeit ist das *Anreichern* der Java Semantik, so dass nicht nur konkreter Java Code behandelt werden kann, sondern schematischer Java Code genutzt werden kann. Das ist nötig, da die Regeln die wir validieren wollen, schematischen Java Code enthalten.

Unser Ziel ist es, einen großen Teil der *Codetransformationstaclets* mittels der in Rewriting Logic gegebenen Semantik für Java zu validieren, sowie die *aussagenlogischen Taclets* mit Rewriting Logic, implementiert in Maude, zu validieren.

1.1 KeY

Diese Diplomarbeit wurde im Rahmen des KeY-Projektes [ABB+05] erstellt. Das KeY-Projekt hat als Ziel, Hilfsmittel für die formale Spezifikation und Verifikation von Software zu entwickeln. Das Hauptaugenmerk des Projektes liegt auf der Entwicklung eines integrierten, interaktiven und automatischen Beweisers für die Verifikation von JavaCard-Programmen, der auf einer dy-

namischen Logik für JavaCard [Bec00, Bec01] basiert. JavaCard ist eine Variante von Java für SmartCard-Software, die einige Teile von Java ausspart, insbesondere Nebenläufigkeit.

Diese Diplomarbeit befasst sich mit dem (automatischen) Nachweis der Korrektheit eines Teiles der zentralen Beweisregeln des im KeY-Projekt entwickelten Beweisers, welcher auf einem Sequenzenkalkül basiert. Diese Regeln sind als Taclets implementiert und sind eher als leichte Taktiken zu sehen, denn nur als Regeln. Taclets sind einfach zu erlernen und ihre wesentlichen Bestandteile sind überschaubar. Für uns relevant sind die sogenannten *find* und *replacewith* Teile. Sie sind wie folgt zu charakterisieren:

- *find*: Der *find* Teil ist eine schematische Beschreibung desjenigen Teils der Sequenz, welcher vom Taclet geändert wird. Dafür muss eine passende Instantiierung des *find* Teils existieren.
- *replacewith*: Der *replacewith* Teil ersetzt das, worauf der *find* Teil gepasst hat. Dabei muss dieser entsprechend der Instantiierung des *find* Teil instantiiert werden.

Taclets beinhalten sogenannte Schema-Variablen, diese sind in Instantiierungen maßgeblich zu berücksichtigen. Schema-Variablen sind typisierte Variablen, die gemäß ihres Typs zum Beispiel auf ganze JavaCard Ausdrücke instantiiert werden können oder auch nur eine einzelne Variable ersetzen können.

Die Taclets, mit denen wir uns beschäftigen werden, sind diejenigen, die nur auf Code Ebene arbeiten, weshalb wir sie als Codetransformationstaclets (CTT) bezeichnen. Diese Taclets nehmen keine Veränderungen an umgebenden Formeln vor, sondern sind auf Code-Stücke, sowohl im *find* als auch im *replacewith* Teil eingeschränkt. Mehr über Taclets im allgemeinen findet man in [BGH$^+$04].

Wir werden die Validierung von CTTs mit Hilfe einer Rewriting Logic Semantik für Java vornehmen, welche in Maude gegeben ist. Maude ist eine auf Rewriting Logic basierende Sprache, die in Abschnitt 1.2 erläutert wird.

In einem weiteren Teil der Arbeit beschäftigen wir uns mit Taclets für Aussagenlogik. Für diese brauchen wir zwei weitere Teile:

- *if*, der *if* Teil muss, genau wie der *find* Teil instantiiert, in der Sequenz zu finden sein.
- *goaltemplate*, der *goaltemplate* Teil besteht aus eimem *add* Teil und

einem *replacewith* Teil, wobei der *add* Teil optional ist und in unserer Arbeit nicht auftreten wird. Also wird aus jedem *goaltemplate* Teil ein *replacewith* Teil. Es können mehrere *goaltemplate* Teile auftreten, was die Sequenz in mehrere Beweisziele aufspaltet.

1.2 Rewriting Logic

Rewriting Logic [MOM02] hat als zentrale Axiome sogenannte Rewrite-Regeln der Form $t \rightarrow t'$ wobei t und t' Terme einer gegebenen Sprache sind. Solche Rewrite-Regeln können auf 2 Ebenen betrachtet werden, einmal logisch, das andere mal berechnungstechnisch. Ein Beispiel hierzu befindet sich in [MOM99]. Rewrite-Theorien, welche Instantiierungen der Rewriting Logic mit konkreten Rewrite-Regeln darstellen, werden unter anderem in [MOM02] erläutert. Rewriting Logic kann als logisches und semantisches Rahmenwerk eingesetzt werden (siehe [MOM00]).

Die Maude Sprache basiert auf Rewriting Logic und ermöglicht es, Rewrite-Theorien darzustellen, mit deren Hilfe auch zum Beispiel eine ausführbare Spezifikation der Java Semantik möglich ist. Eine Einführung in Maude bietet das Maude Manual [CDE+00]. Die ausführbare Spezifikation der Java Semantik stellt unmittelbar einen Interpreter für Java dar. Diese Java Semantik wird genutzt in [FCMR04] und [FMR04], aber nicht erläutert. Um die Semantik für Java besser zu verstehen, bietet es sich an, zuvor eine Spezifikation der Semantik einer kleineren Sprache, die ähnlich zu CaML ist, anzusehen. Diese ist in [MR04] näher beschrieben.

Für oben erwähnten Nachweis der Korrektheit werden wir die Code-Teile von CTTs in der Maude Java Semantik (MJS) ausführen und die Ergebnisse vergleichen. Dazu werden beide Programmteile in gleichen Zuständen gestartet. Diese Zustände der MJS enthalten den Speicher, die Umgebungen und den Code neben anderen syntaktisch relevanten Dingen. Wir bezeichnen diesen Gesamtzustand als *Konfiguration*.

1.3 Ziel der Arbeit

Das Ziel der Arbeit ist es, die Korrektheit von Taclets, mit Hilfe von Maude, zu untersuchen. Dazu betrachten wir zwei Punkte:

- Wir validieren CTTs mittels der MJS. Dafür werden beide Code-Stücke ausgeführt, müssen allerdings vom vorliegenden schematischen Code umgewandelt werden in einen konkreten, aber generischen, Code, wie später beschrieben, wofür auch Unterstützung von Seiten der Semantik notwendig ist.
- Wir validieren die axiomatischen aussagenlogischen Taclets mit Hilfe von Maude.

Wir wollen mit dem ersten Punkt sicherstellen, dass die Taclets exakt der Java Language Specification [GJSB00] (JLS) entsprechen. Bis dato wurden die Taclets nur informell gegenüber der JLS geprüft. Nun können wir die Code-Segmente in der MJS vergleichen und sind damit in der Lage, Fehler zu entdecken.

2 Korrektheit von Codetransformationstaclets

Pro CTT haben wir zwei Code-Segmente, Π im *find*-Teil und Π' im *replacewith*-Teil. Wir wollen nun zeigen, dass diese äquivalent sind. Dafür müssen wir die Gleichheit des Ergebnisses der beiden Programmstücke für jeden beliebigen Startzustand s zeigen. Mit einer an Structural Operational Semantics [NN92] (SOS) angelehnten Notation wollen wir also $< \Pi, s > == < \Pi', s >$ für alle s zeigen. Wir erzeugen hierzu statt aller Startzustände s einen generischen Startzustand, welcher alle Möglichkeiten repräsentiert. Dies wird nicht immer gehen, jedoch können wir nach genauerer Betrachtung insbesondere der Schema-Variablen immer mit einer akzeptablen Menge an generischen Startzuständen den Beweis antreten. Diese Menge an generischen Startzuständen reicht aus, um alle möglichen Startzustände zu simulieren. Dann muss die Äquivalenz natürlich für all diese generischen Startzustände gelten. Die Abarbeitung der Programme, beschrieben durch obige Notation, geschieht in Maude mit Hilfe der MJS.

Die notwendigen Fallunterscheidungen werden durch die in den Programmen auftretenden Schema-Variablen bedingt. Hierfür ist eine genaue Betrachtung jedes einzelnen Schema-Variablen Typs angebracht. Für jeden Schema-Variablen-Typ gibt es verschiedene (sich teils überlappende) Instantiierungsmöglichkeiten. All diesen muss bei der Erstellung der generischen Startkonfigurationen Rechnung getragen werden. Insbesondere dann, wenn, wie üblich, mehr als eine Schema-Variable in einem Code-Stück auftritt, müssen alle Kombinationen aller Möglichkeiten aller Schema-Variablen berücksichtigt werden.

Im Anschluss ist noch zu vermerken, dass die obige Prüfung auf Äquivalenz einige Einschränkungen hat. In der Taclet-Sprache gibt es Ausdrücke, die verlangen, dass *neue Variablen* eingeführt werden. Diese werden nur für den *replacewith*-Teil erzeugt, um dort zum Beispiel als Zwischenspeicher zu dienen. Diese neuen Variablen sind im Code des *find*-Teils natürlich nicht vorhanden und machen somit eine Gleichheit der Ergebnisse unmöglich. Deshalb betrachten wir die Äquivalenz modulo neuer Variablen, d.h. wir verwerfen al-

le neuen Variablen, was auch vollkommen in Ordnung ist, denn diese neuen Variablen können hinterher nicht wieder auftreten, da sie ja nach Definition der Taclet-Sprache neu für den vorhandenen Code sind. Um dies in der Praxis zu bewerkstelligen existieren von uns hinzugefügte Hilfs-Operatoren in der MJS. Das Verfahren ist in gewissem Sinne korrekt.

3 Anreichern der Semantik

Das Anreichern der Semantik ist nötig, da wir zwei Code-Segmente vergleichen wollen, welche als Schemata gegeben sind, und dafür eine Semantik für konkreten Code nutzen wollen. Einführend erweitern wir die Semantik um einen pause-Operator, welcher uns ermöglicht, den Programmlauf an einer Stelle anzuhalten und die dort vorhandene Konfiguration genauer zu betrachten.

Zuerst betrachten wir die Behandlung von *neuen Variablen*, welche dazugehörige *neue Plätze* im Speicher bekommen und für die wir neue Sorten einführen, die die Unterscheidung von alten und neuen Elementen erleichtert. Des weiteren führen wir in diesem Kapitel Hilfsmittel ein, welche der Beschreibung und Entscheidung der Äquivalenz modulo neuer Variablen dienen. Hierbei werden am Ende der Abarbeitung alle neuen Variablen, samt dazugehörigen neuen Speicherplätzen und den dortigen Werten, verworfen. Aufgrund des Aufbaus der verwendeten Hilfsmittel gibt es keine Probleme, was die Verwechslung von alten und neuen Konfigurationsteilen angeht.

Ausdrücke können Seiteneffekte hervorrufen und liefern, wenn sie terminieren und keine Ausnahme verursachen, ein Resultat. Wir haben es hier nun mit schematischen Ausdrücken zu tun, von denen wir nicht wissen, was für Seiteneffekte sie nach sich ziehen. Um dies zu behandeln muss die MJS in einigen Punkten angepasst werden, so dass solche Ausdrücke (und ihre Effekte) dargestellt werden können. Hierbei kommen etliche Hilfs-Konstrukte zum Einsatz, insbesondere nutzen wir *erweiterte bedingte Werte*, welche in ihrer Bedingung den Abgleich eines Speicherplatzes mit einer Liste von Speicherplätzen zulassen und, falls eine Übereinstimmung vorliegt, aus einer Liste von Werten den an der entsprechenden Stelle stehenden Wert zum Ergebnis machen. Dies ermöglicht uns die Darstellung von Seiteneffekten in der Form, dass anstelle des ursprünglichen Wertes im Speicher nun ein solcher erweiterter bedingter Wert steht, den wir allerdings nicht auswerten müssen, da wir ja nur zwei Programmstücke miteinander vergleichen wollen.

Das Einbringen dieser erweiterten bedingten Werte in den Speicher bedarf einiger Aufmerksamkeit, da Werte im Speicher nur durch Werte des gleichen Typs überschrieben werden dürfen und insbesondere im Fall von Objektreferenzen der statische Typ nicht verändert werden darf. Das Einbringen erweiterter bedingter Werte findet dann statt, wenn ein Ausdruck mit möglichem Seiteneffekt ausgeführt wird, um eben jenen Seiteneffekt darzustellen.

Um das Auftreten zweier möglicherweise seiteneffektbehafteter Ausdrücke zu vergleichen, betrachten wir noch zusätzlich Momentaufnahmen ("Snapshots") von Teilen der Konfiguration. Wenn ein Ausdruck in zwei verschiedenen Programmläufen ausgeführt wird, aber jeweils der relevante Teil, der in der Momentaufnahme festgehalten wird, zum Ausführungszeitpunkt identisch ist, ist sowohl der Seiteneffekt, als auch das Ergebnis der beiden Ausführungen das gleiche, falls der Ausdruck terminiert. Falls der Ausdruck nicht terminiert, gilt das in beiden Fällen

4 Veränderungen an der Semantik

Weitere Änderungen an der MJS sind nötig, um Fehler zu beheben, weitere Besonderheiten der Tacletsprache einzubauen sowie Vereinfachungen für unsere Art der Verwendung zu ermöglichen.

Fehler, die korrigiert worden sind, sind unter anderem die Auswertung von Konjunktionen und Disjunktionen, Fehler bei der Erstellung von Arrays, Probleme beim Aufruf von Methoden von Oberklassen auf Objekten einer Unterklasse und fehlerhafte interne Vorgehensweisen, die die gesamte Abarbeitung gestoppt haben. Zudem war die Semantik von Blöcken nicht korrekt erfasst.

Hinzugenommen wurden Typtests bei Typumwandlungen. Vorher wurden Typumwandlungen ohne Test einfach durchgeführt. Bei Zuweisungen kamen ebenfalls Typtests hinzu. Daher passiert es nun nicht mehr, dass eine als Typ `A` deklarierte Objektreferenz plötzlich den statischen Typ `B` besitzt.

Außerdem wurde ein Konstrukt der Taclet-Sprache hinzugefügt, nämlich simultane Updates. Diese können nun als Beginn eines Code-Stücks auftreten und haben die Semantik wie die Taclet-Sprache es verlangt.

Leider konnten etliche fehlende Konstrukte im Rahmen dieser Diplomarbeit nicht ergänzt werden, wovon uns das Fehlen von Ausnahmen ("Exceptions") besonders getroffen hat. Die fehlenden Konstrukte sind *throw*, *try-catch-finally*, *switch*, *break*, *continue*, *conditional expression*. *Interfaces* gibt es nicht und *Methodenüberladung* ist nicht möglich. Es gibt zudem keine Standard-Konstruktoren, diese müssen von Hand für jede Klasse angegeben werden und dabei ihre Superklasse aufrufen.

5 Erzeugung der Konfigurationen

Die Erzeugung der erwähnten Startkonfigurationen geschieht in einem zweistufigen Prozess. Zuerst werden mittels eines Java Programms, welches in KeY integriert wurde, die zu prüfenden Taclets eingelesen und in die relevanten Code-Teile zerlegt. Zudem wird festgestellt, welche Variablen es gibt, und diese werden in die Konfiguration-Erzeugungs-Strings integriert. Diese Erzeugungs-Strings werden in eine Datei gespeichert, von Maude eingelesen, und daraus wird die eigentliche Startkonfiguration erzeugt. Direkt im Anschluss werden dann die beiden Code-Teile in ihren jeweiligen Konfigurationen ausgeführt und die Resultate verglichen.

In der MJS ist hierfür eine Schnittstelle implementiert worden, welche Strings obiger Form, mit Code-Teil und hinzuzufügenden Variablen, genauer angibt und ihre Verarbeitung übernimmt.

Im Java-basierten Teil findet noch ein weiterer wichtiger Schritt statt, nämlich die Erzeugung aller möglicher Kombinationen der Fälle für die Schema-Variablen. Hierbei können je nach Taclet mehrere hundert Startkonfigurationen erstellt werden, was aber durch die automatische Erzeugung und Verarbeitung kein Problem darstellt. Durch diese Menge an Startzuständen werden, wie oben erwähnt, alle möglichen Startzustände dargestellt.

6 Aussagenlogische Taclets in Rewriting Logic

Hier werden die aussagenlogischen Taclets des KeY-Systems in Rewriting Logic nachgebaut, also imitiert, um dort mit einer Semantik für Aussagenlogik validiert zu werden. Dieser Ansatz funktioniert, allerdings nur durch das Ausnutzen der Tatsache, dass wir es mit Aussagenlogik zu tun haben. Erweiterbarkeit des Ansatzes auf dynamische Logik für JavaCard ist nicht einfach herzustellen, wenn es überhaupt möglich sein sollte. Die hier imitierten Taclets konnten dann auch vollständig als korrekt gezeigt werden. Teile der hier genutzten Notation wurde von [CM00] inspiriert.

Zudem verfolgen wir die Idee, die Taclets nicht nur durch Regeln einer Rewriting Logic zu imitieren, sondern sie einzubetten, d.h. Möglichkeiten anzubieten, so dass Taclets so normal wie möglich eingegeben werden können und ein Anwendungsmechanismus vorhanden ist. Mit gewissen Einschränkungen funktioniert dies. Beweistechnisch ist hier das Problem, Terme über Schema-Variablen zu erstellen. Wegen des hohen technischen Aufwands ist das aber nicht weiter verfolgt worden.

7 Ergebnis

Wir haben in dieser Arbeit aufgezeigt, wie KeY Codetransformationstaclets (CTT) mit Hilfe der Maude Java Semantik (MJS) validiert werden können. Wir haben ebenso gezeigt, wie aussagenlogische Taclets mit der Hilfe von Maude validiert werden können.

Unser Hauptaugenmerk lag dabei auf den CTTs. Von den ungefähr 350 Taclets, die sich mit Java Code befassen, sind ca. 140 CTTs, also 40%. Wir haben bereits 55 dieser Taclets validiert. Einige weitere Taclets sollten mit kleinen technischen Änderungen an unserer Implementierung validierbar sein. Die Anzahl der noch offenen CTTs kommt daher, dass KeY Meta-Konstrukte genutzt werden, die sich nicht einfach in die MJS übertragen lassen und zudem die MJS nicht alle Features von Java vollständig abdeckt.

Der Großteil unserer Arbeit reicherte eine Semantik für gewöhnlichen Java Code an, so dass es eine Semantik für schematischen Java Code wurde. Das war notwendig, da die Code-Stücke, welche wir von den CTTs her betrachten, aus schematischem Java Code bestehen. Daraus ergab sich eine Menge von Fragen und Problemen welche im Laufe der Arbeit gelöst wurden.

Die Veröffentlichung [MR04] führt eine auf Rewriting Logic basierende Semantik einer CaML-ähnlichen Sprache ein. Hier war es für uns wichtig, die Prinzipien einer auf Rewriting Logic basierenden Semantik für Programmiersprachen, welche Continuations verwendet, zu verstehen. Die Veröffentlichung war dadurch ein sehr guter Anfang um die MJS zu verstehen, für die, soweit wir wissen, keine genauere Dokumentation zur Verfügung steht. In der Veröffentlichung war die MJS auch angegeben als eine Semantik von Java, allerdings wurde nichts über die Vollständigkeit und Korrektheit gesagt. Wir haben im Laufe unserer Arbeit festgestellt, dass es einige Fehler in der MJS gibt und sie nicht alle Features von Java abdeckt.

Die Verfügbarkeit der MJS war zentral für unsere Arbeit. Trotz aller Einschränkungen, was Korrektheit und vollständige Abdeckung der Features angeht, war die MJS eine grosse Hilfe. Um sie vernünftig nutzen zu können,

mussten wir allerdings einiges leisten. Die meisten Fehler, und deren Bereinigung, waren alles andere als offensichtlich. Einige Fehler konnten behoben werden, während andere so zentral eingebaut sind, dass in der kurzen, uns zur Verfügung stehenden Zeit, diese Fehler nicht zu beheben waren. Außerdem haben wir noch einige Fähigkeiten zur MJS hinzugefügt, die uns bei unserer Arbeit geholfen haben.

Unser anderer Ansatz, der sich mit aussagenlogischen Taclets befasst, war in der Lage, sämtliche axiomatischen aussagenlogischen Taclets, die im KeY System enthalten sind, zu validieren.

Als zukünftige Arbeit schlagen wir vor, einige technische Verbesserungen vorzunehmen sowie die MJS zu vervollständigen. Wenn das erledigt ist, könnte man die Probleme in Angriff nehmen, die bei Entscheidungen mit boolschen Variablen auftreten. Hierbei ist das Problem, dass immer wenn ein `if` oder ähnlicher Operator auftritt, der einen boolschen Wert, also wahr oder falsch, als Argument benötigt und nur eine generische uninterpretierte Konstante vorliegt, die Ausführung des Codes nicht fortgesetzt werden kann. Hierfür könnte man die Ausführung in zwei Teile aufspalten und die zwei kleineren Teilprobleme betrachten.

Die Validierung von Taclets mit Meta-Konstrukten ist eine ebenso offene Frage. Es könnte möglich sein, die Tatsache auszunutzen, dass die Meta-Konstrukte als Java Code implementiert sind. Uns ist noch keine Lösung hierfür bekannt, wobei als Voraussetzung die Vollständigkeit der MJS nötig ist.

Wir betrachten diese Arbeit als Beitrag im Bereich der (teil-)automatischen Beweiser und ihrer Korrektheit. Wir haben insbesondere im Rahmen des KeY Systems, betreffend die Korrektheit, einen Schritt nach vorne gemacht, dadurch, dass unser Ansatz einen Teil der Regeln validiert hat, die dem Beweiser zugrunde liegen.

Literaturverzeichnis

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Bec00] Bernhard Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BGH+04] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. SRI International, 2000. Available from `http://maude.cs.uiuc.edu/papers/`.

[CM00] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[FCMR04] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal analysis of java programs in javafan. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[FMR04] Azadeh Farzan, José Meseguer, and Grigore Rosu. Formal jvm code analysis in javafan. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[MOM99] N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhofer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

[MOM00] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[MR04] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proceedings of the IJCAR'04, Cork, Ireland*, volume 3097, pages 1–44. Springer-Verlag LNCS, July 2004.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.